Math Part of Sigmoid Neuron

medium.com/@manveetdn/notes-on-math-part-of-sigmoid-neuron-padhai-onefourthlabs-course-a-first-course-on-deep-4e9af854ff56

> *Disclaimer: This is notes on* **"Math Part of Sigmoid Neuron"** *Lesson (PadhAI* ***onefourthlabs*** *course "A First Course on Deep Learning")*

Learning Algorithm:

We know to update the weight of w using the perticualr formulae.

w = w+ **η**Δw **[η is a small value]**

What we need mainly after the update is

Loss(w)> Loss(w+ ηΔw)

> The loss should decrease after we update the value of w.

We will do all these using Taylor series.

Taylor Series:

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 \\ &\quad + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f''''(x_0)}{4!}(x - x_0)^4 + \cdots \\ &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \end{aligned}$$

This is the formulae of the Taylor series.

Taylor Series

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2!} f''(x)\Delta x^2 + \frac{1}{3!} f'''(x).\Delta x^3 + \ldots$$

This is what we use in deep learning.

It says that

If you have a function and if you know the value of it at any point then its value at a new point which s closer to the older point can be given by above formula.

$$\underbrace{f(x+\Delta x)}_{\text{newpoint}} = \underbrace{f(x)}_{\text{before point}} + \underbrace{\frac{1}{1!} f'(x)\Delta x + \frac{1}{2!} f''(x)\Delta x^2 + \frac{1}{3!} f'''(x)\Delta x^3 + \dots}_{\text{adding before with this}}$$

$$f(x+\Delta x) = f(x) + \underbrace{\left[\frac{1}{1!} f'(x)\Delta x + \frac{1}{2!} f''(x)\Delta x^2 + \frac{1}{3!} f'''(x)\Delta x^3 \dots\right]}_{-ve}$$

If

then $f(x+\Delta x) < f(x)$

New loss < old loss

$\therefore$ we need to find Δx which turns whole [ ] value to negative

If $f(x) = x^3$

$f'(x) = 3x^2$

$f''(x) = 6x$

$f'''(x) = 6$

$f''''(x) = 0$

Thathow we use talyor series.

We know that $x^3 = 27$ we can write that as $f(x) = x^3$

What is the value of $f(27+0.000001)^3$

$$x^3 = 27$$

$$f(27 + 0.0001)^3$$

$$x^3 + 3x^2 \Delta x + \frac{1}{2!} 6x (\Delta x)^2 + \frac{1}{3!} 6 (\Delta x)^3 + 0 + 0 + \cdots$$

$$= 27 + 27(0.0001) + \frac{1}{2!} 18 (0.0001)^2 + \frac{1}{3!} 6 (0.0001)^3 + 0$$

$$= 27 + 27(0.0001) + 9(0.0001)^2 + (0.0001)^3$$

Applying Taylor series.

In the same way we use if we use it fr the loss function while applying it for the w or b then we use it as follows.

$$\mathcal{L}(w + \Delta w) = \mathcal{L}(w) + \left[\mathcal{L}'(w) \cdot \Delta w + \left(\mathcal{L}''(w) (\Delta w)^2\right) \frac{1}{2!} + \frac{1}{3!} \left(\mathcal{L}'''(w) (\Delta w)^3\right) + \cdots\right.$$

$$\therefore L(w + \Delta w) = L(w) + \underbrace{\left[L'(w) \Delta w + \frac{1}{2!} L''(w) \Delta w^2 + \frac{1}{3!} L'''(w) (\Delta w)^3 + \cdots\right]}_{\text{If -ve then only}}$$

$$\Rightarrow L(w + \Delta w) < L(w)$$

Applying it for the loss when updating w.

Actually It is

$$L(w,b) > L(w+\eta\Delta w,\ b+\eta\Delta b)$$

before change   After change

let $\theta = [w,b]$ $\theta + \Delta\theta$

$$L(\theta) > L(\theta+\eta\Delta\theta) \qquad \begin{bmatrix} w \\ b \end{bmatrix} + \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix}$$

The before taylor series was for scalar case and not for vector

∴ Taylor series for vector case :-

$$L(\theta+\eta u) = L(\theta) + \eta * u^T \nabla_\theta L(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L(\theta) u + \cdots$$

$$L(\theta+\eta u) = \underbrace{L(\theta)}_{\text{old}} + \underbrace{\left[\eta * u^T \nabla_\theta L(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L(\theta) u + \cdots\right]}_{\text{new}}$$

we need change vector that this quantity will be -ve

Such that

$$\underbrace{L(\theta+\eta u)}_{\text{new loss}} < \underbrace{L(\theta)}_{\text{old loss}}$$

Finally we follow like this t suffice our need of decreasing the loss function value.

We have the equation

$$L(\theta+\eta\mu) = L(\theta) + \eta*\mu^T \nabla_\theta L(\theta) + \left[\frac{\eta^2}{2!} * \mu^T \nabla^2 L(\theta) u + \dots\right]$$

The whole equation is so complex so we get rid of η as it is very small η^2 & η^3 & η^4 will also be very negligable

$$\therefore L(\theta+\eta u) = \underset{L(\theta)}{a} + \underset{\eta*\mu^T\nabla_\theta L(\theta)}{b} + [c]$$

$[c]$ → very small remain part in above brackets

∴ We can write the above equation as

$$L(\theta+\eta u) \approx L(\theta) + \underbrace{\eta * u^T \nabla_\theta L(\theta)}$$

so we need this quantity to be negative such that old loss > new loss

$\nabla_\theta L(\theta)$ is the first derivative

$$f(w,b) = w^3 + b^3$$

here we take partial derivable as w, b both are variables

②

$$\therefore \frac{\partial f(w,b)}{\partial w} = \frac{\partial}{\partial w}(w^3) + \frac{\partial}{\partial w}(b^2)$$

here in the case b is constant when we take partial derivative of w

$$\frac{\partial f(w,b)}{\partial w} = 3w^2 + 0$$

$$\frac{\partial f(w,b)}{\partial w} = 3w^2$$ partial derivative w.r.t 'w' $= 3w^2$

$||^{ly}$ partial derivative with b

$$\frac{\partial (f(w,b))}{\partial b} = \frac{\partial}{\partial b}(w^3) + \frac{\partial}{\partial w}(b^2)$$

Now w is constant

$$\frac{\partial f(w,b)}{\partial b} = 0 + 2b$$

$$\frac{\partial f(w,b)}{\partial b} = 2b$$

∴ This is partial derivate w.r.t b

$$\begin{bmatrix} \frac{\partial f}{\partial w} \\ \frac{\partial f}{\partial b} \end{bmatrix} = \begin{bmatrix} 3w^2 \\ 2b \end{bmatrix}$$

we put these partial derivatives in a vector what we get here is gradient (∇) of function f depending on θ

∴ $\nabla_\theta f(\theta)$ gradient of function depending on θ.

$\nabla_\theta f(\theta)$ - This is the gradient of function $f(\theta)$ with respect to θ

Which is $\begin{bmatrix} \frac{\partial f}{\partial w} \\ \frac{\partial f}{\partial b} \end{bmatrix} = \begin{bmatrix} 3w^2 \\ 2b \end{bmatrix}$ put in vector form is the gradient

$\cancel{L(\theta+\eta u)}$ $\cancel{L(\theta) + \eta * \nabla_\theta L(\theta)}$

$$\underbrace{L(\theta+\eta u)}_{\mathbb{R}} = \underbrace{L(\theta)}_{\mathbb{R}} + \underset{\mathbb{R}}{\eta} * \mu^T \nabla_\theta L(\theta)$$

$$\mu = \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix} \quad \mu^T = [\Delta w \ \Delta b]$$

$$\nabla_\theta L(\theta) = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}$$

$$\mu^T * \nabla_\theta L(\theta) = \underbrace{[\Delta w \ \Delta b] \cdot \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}}_{\text{dot product}}$$

dot product → $\mathbb{R}$ number itself

$$L(\theta+\eta u) = L(\theta) + [\eta * \mu^T \nabla_\theta L(\theta)]$$

↓ We should find a u such that this whole is −ve

$$L(\theta+\eta u) = L(\theta) + \eta * u^T \nabla_\theta L(\theta) \quad —①$$

$$L(\theta+\eta u) - L(\theta) = \eta * \mu^T \nabla_\theta L(\theta) \quad →②$$

Now we want is

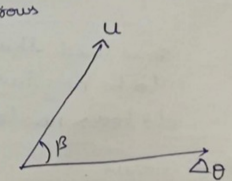
$$L(\theta+\eta u) - L(\theta) < 0 \quad ③$$ [i.e., if new loss is less than prev loss]

This implies comparing ② & ③

$$u^T \nabla_\theta L(\theta) < 0$$

$u^T \nabla_\theta L(\theta)$ → $\mathbb{R}^2$ $\quad \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix} \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}$

↓ Matrix in 2 rows $\mathbb{R}^2$



Let μ & $\Delta\theta$ be the vectors the cos of angle

$$\cos\beta = \frac{\mu^T \nabla_\theta L(\theta)}{||u|| \, ||\nabla_\theta L(\theta)||}$$

As we came up with

$$\cos\beta = \frac{u^T \nabla_\theta L(\theta)}{||u|| * ||\nabla_\theta L(\theta)||}$$

$$\therefore \quad -1 \le \cos\beta = \frac{\mu^T \nabla_\theta L(\theta)}{||u|| * ||\nabla_\theta L(\theta)||} \le 1$$

multiply with $k = ||u|| * ||\nabla_\theta L(\theta)||$

$$\Rightarrow \quad -k \le \cos\beta = \mu^T \nabla_\theta L(\theta) \le k$$

$$-k \le u^T \nabla_\theta L(\theta) \le k$$

Thus $L(\theta + \eta u) - L(\theta) = u^T \nabla_\theta L(\theta) = k * \cos\beta$

will be more negative when $\cos(\beta) = -1$

i.e., when β is 180°

Gradient Descent rule:

1. The direction u that we intend t move in should be 180 w.r.t gradient
2. In other words, move in direction opposite to the gradient.

Parameter Update Rule

$$w_{t+1} = w_t - \eta \Delta w_t$$

$$b_{t+1} = b_t - \eta \Delta b_t$$

where $\Delta w_t = \dfrac{\partial L(w,b)}{\partial w}$ at $w = w_t, b = b_t$

$\Delta b_t = \dfrac{\partial L(w,b)}{\partial b}$ at $w = w_t, b = b_t$

∴ Learning Algorithm

Initialize

w, b

Iterate over data

compute $\hat{y}$

compute $L(w,b)$

$$w_{t+1} = w_t - \eta \boxed{\Delta w_t} \rightarrow \frac{\partial L}{\partial w}$$

$$b_{t+1} = b_t - \eta \boxed{\Delta b_t} \rightarrow \frac{\partial L}{\partial b}$$

computegrad(L, θ)

till satisfied

we will update the weights as required and then we will compute loss and when the loss is minimum then we will finalize the values means 100 iteration or 1000 iteration or upto particular threshold

Like this we use the Gradient descent rule.

$$L = \frac{1}{5}\sum_{i=1}^{5}(f(x_i) - y_i)^2$$

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{5}\sum_{i=1}^{5}(f(x_i) - y_i)\right]$$

$$\Delta w = \frac{\partial L}{\partial w} = \frac{1}{5}\sum_{i=1}^{5}\frac{\partial}{\partial w}(f(x_i) - y_i)$$

let one of the term ins terms be

$$\nabla w = \frac{\partial}{\partial w}\left[\frac{1}{2} * (f(x) - y)^2\right]$$

$$= \frac{1}{2} * \left[2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)\right]$$

$$= (f(x) - y) * \frac{\partial}{\partial w}(f(x))$$

here $f(x) = \frac{1}{1+e^{-wx+b}}$

$$= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right)$$

$$\frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right)$$

let $1 + e^{-(wx+b)} = P$

$$\therefore \frac{\partial}{\partial w}\left(\frac{1}{P}\right) = -\frac{1}{P^2}\frac{\partial P}{\partial w}$$

$$= -\frac{1}{(1+e^{-(wx+b)})^2} * \frac{\partial}{\partial w}\left(e^{-(wx+b)}\right)$$

let $z = -(wx+b)$ $\Rightarrow -\frac{1}{(1+e^{-(wx+b)})^2} * e^z \frac{dz}{dw}$

$$= -\frac{1}{(1+e^{-(wx+b)})^2} * \left(e^{-(wx+b)}\right)\frac{\partial}{\partial w}(-(wx+b))$$

$$\boxed{\therefore \frac{\partial}{\partial w}(b) = 0,\ \frac{\partial}{\partial w}w = 1}$$

$$= \frac{-1}{(1+e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})} * (-x)$$

$$= \frac{x\, e^{-(wx+b)}}{(1+e^{-(wx+b)})} * \frac{1}{(1+e^{-(wx+b)})}$$

$$= \underbrace{\frac{1}{(1+e^{-(wx+b)})}}_{f(x)} * \frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})} * x$$

$$\therefore \frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right) = \underbrace{\frac{1}{(1+e^{-(wx+b)})}}_{f(x)} * \underbrace{\frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})}}_{(1-f(x))} * \underbrace{x}_{x}$$

$$= f(x) * (1 - f(x)) * x$$

$$\therefore \nabla w = (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right)$$

$$= (f(x) - y) * (f(x) * (1 - f(x)) * x)$$

This solving using gradient descent.

> **Finally after doing all the math we come up with the final formulae for all the updating the values of the w and b i.e., Δw, Δb as shown below.**

Now as we have for one value of w

$$\nabla w = (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right)$$

$$\nabla w = (f(x) - y) * f(x) * ((1 - f(x))) * x$$

replace all s values with this expression

$$\Delta w = \sum_{i=1}^{s} (f(x_i) - y_i) * f(x_i)) * (1 - f(x_i)) * (1 - f(x_i)) * x_i$$

$$\Delta b = \sum_{i=1}^{s} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

This is the final formulae to calculate Δw, Δb

So the code we write for sigmoid neuron model involves all these as below.

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x):
   #sigmoid with parameters w, b
   return 1.0 / (1.0 + np.exp(-(w*x + b))

def error(w, b):
     err = 0.0
     for x, y in zip(X, Y):
        fx = f(w, b, x)
        err += 0.5* (fx - y) ** 2
     return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x


def do_gradient_descent():
    w, b, eta = -2, -2, 1.0
    max_epochs = 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

Code for sigmoid neuron model

Finally the code

step 1 Initializing

x = [0.5, 0.2]

y = [0.2, 0.9]

step 2 def f(w, b, x):

return 1.0/(1.0 + np.exp(-(w*x + b)))

This is nothing but $\frac{1}{1+e^{-(wx+b)}}$ np.exp(x) is $e^{(x)}$

This is sigmoid function code

step 3 def error(w, b):

err = 0.0

for x, y in zip (X, Y):

fx = f(w, b, x)

err += 0.5*(fx-y)**2

return err

here we are computing the errors firstly we initialize it to zero then iterate through x, y data and using sigmoid we will find the value and store in fx and again

err += 0.5*(fx-y)**2 is the square error loss

0.5*(fx-y)**2 $= \frac{1}{2}\cdot(fx-y)^2 = \frac{1}{2}(\text{pred value} - \text{true value})^2$

like the for the values we compute square error loss

step 4 :- def grad_b(w, b, x, y):

fx = f(w, b, x)

return (fx-y) * fx * (1-fx)

here we are calculating gradient of b which we proved mathematically before with this we will find Δb

step 5 :- def grad_w(w, b, x, y):

fx = f(w, b, x)

return (fx-y) * fx * (1-fx)

Similar to gradient of b we will find gradient of w here i.e., Δw

step 6 :- def do_gradient_descent():

Initializing values → w, b, eta = -2, -2, 1.0

no of ephocs → max_epochs = 1000

Iterating in range of ephocs → for i in range(max_epochs):

$\Delta w, \Delta b = 0$ → dw, db = 0, 0

Iterating through data → for x, y in zip(X, Y):

Find Δw & Δb $\begin{cases} dw += grad\_w(w,b,x,y) \\ db += grad\_b(w,b,x,y) \end{cases}$

updating with w as $w - \eta\Delta w$ b as $b - \eta\Delta b$ $\begin{cases} w = w - eta * dw \\ b = b - eta * db \end{cases}$

step 6 is the main function part firstly we will initialize w, b, eta to some values and we will declare max no of epochs And we will iterate through data in range of epochs we will be updating value of dw & db with the gradients functions and then for each epoch we will be updating

w = w - eta*dw

b = b - eta*db

like that decrease with eta*Δw and eta*Δb with a though to decrease loss than before one

5

If we have Two parameters :-

So far we came across only one parameter called x

If we have multiple parameters like then we will take each parameter as x_i if we have parameter then we will assign it is x_1, x_2 if 3. x_1, x_2, x_3 if n $x_1, x_2, x_3 ... x_n$.

like the $z = wx + b$ for one parameter

where $\frac{1}{1+e^{-z}}$

Now for multiple parameter

$$z = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + \dots + b$$

$$\hat{y} = \frac{1}{1+e^{-(wx+b)}}$$ one parameter

$$\hat{y} = \frac{1}{1+e^{-(w_1x_1+w_2x_2+w_3x_3+\dots+b)}}$$ many parameters

```python
def grad_w_i(w, b, x, y, i):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x[i]
```

```python
def f(w, b, x):
   #sigmoid with parameters w, b
   return 1.0 / (1.0 + np.exp(-(np.dot(w, x) + b))
```

code for gradient of ‘w’ and sigmoid function after for two and more parameters

Algorithm will be same if we have multiple data

Initialize $w_1, w_2, w_3, \dots, b$

Iterate over data

$w_1 = w_1 - \eta \Delta w_1$

$w_2 = w_2 - \eta \Delta w_2$

$w_3 = w_3 - \eta \Delta w_3$

$\vdots$

$w_n = w_n - \eta \Delta w_n$

till satisfied

But comming to Math part

$\Delta w = \sum_{i=1}^{M} (f(x) - y) * f(x) * (1 - f(x)) * x$

$\Delta w_1 = \sum_{i=1}^{M} (\hat{y} - y) * \hat{y} (1 - \hat{y}) * x_{i1}$

$\Delta w_2 = \sum_{i=1}^{M} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{i2}$

$\vdots$

$\Delta w_j = \sum_{i=1}^{M} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{ij}$

like for every w_j we are going find the values like using the above formula

Now the change in the code is

```
def grad-w-i(w,b,x,y,i):
    fx = f(w,b,x)
    return (fx-y)*fx*(1-fx)*x[i]

def f(w,b,x):
    return 1.0/(1.0+np.exp(-(np.dot(w,x)+b))
```

This piece of code is $1 + e^{-(\sum w_i x_i + b)}$

This whole is $1 / 1 + e^{-(\sum w_i x_i + b)}$.

IIly grad-b-i also should be written and even small changes in the main function will be there

Evaluation:

Here, also the same cse we will evaluate our model on given test data.

Test data contains y and yhat

y = true value , yhat= predicted value

Root mean square error $= \sqrt{\frac{1}{n}\left(\sum_{i=1}^{n}(y-\hat{y})^2\right)}$

Square error loss $= \sum_{i=1}^{n}(y-\hat{y})^2$

Root mean square error and the square error loss are used to calculate the loss value.

Therefore, **RMSE (Root mean square error)** is **mostly used for regression** problems rather than classification. If you force on classification at particular threshold we **need to binaries the outputs.**

> ***Finally, Accuracy is given by the same formulae as the ratio of Number of correct predictions and the Total number of predictions.***

This is a small try, uploading the notes . I believe in "**Sharing knowledge is that best way of developing skills**".Comments will be appreciated. Even small edits can be suggested.

> **Each Applause will be a great encouragement.**

Do follow my medium for more updates……